1. Solving Problems in EUROPA
    1. 1. Partial and Complete Plans
       2. Flaws and Flaw Resolution
       3. Refinement Search

Solving Problems in EUROPA

EUROPA is designed primarily to solve planning and scheduling problems. In classical planning a planning problem is presented in terms of an *initial state* and a *goal state*. It is the role of a *planner* to specify the appropriate details in fleshing out a legal plan to go from one to the other. This is referred to as *state space planning* since the nodes in the search towards a solution vary in the states they result in based on planning operators selected. In contrast, EUROPA formulates a planning problem as an *initial partial plan* which is refined through a series of refinement operations into a final plan that is complete with respect to the requirements of the planner. This approach is referred to as *refinement search* and/or *plan-space planning*, since each node represents a different partial plan obtained by application of a particular refinement operator to a prior partial plan.

This section introduces the concepts of partial and complete plans. It presents in more detail the set of flaws which indicate a gap to be bridged to find a complete plan and methods of flaw resolution for each type of flaw. Finally, a general algorithm for automated planning using chronological backtracking is presented.

Partial and Complete Plans

A domain model deals with the abstractions of classes, predicates and rules. In a *partial plan* these are instantiated as objects,[1] tokens, and rule instances respectively. Informally,[2] a plan is complete if it has no inactive tokens and no unbound variables. While this is a sufficient requirement for completeness, it may not be necessary. For example, suppose that the consumer of a plan (i.e. an execution system) prefers to maintain flexibility in temporal variables. In this case a plan may be complete even if it includes unbound *start*, *end* or *duration* variables. It is also common, even necessary in order for planning to terminate in many cases, that the planner excludes inactive tokens from consideration if they are outside a *planning horizon*.

Figure 1: Planning as a process or iterative refinement of partial plans

Flaws and Flaw Resolution

Figure 1 illustrates an idealized view of the planning process used in EUROPA. The process begins with an initial partial plan. This is typically an allocation of objects and tokens which are either mandatory or rejectable. When such a partial plan is instantiated in EUROPA, it is common that model rules will fire, causing slave tokens to be created. This is indicated in the growth of the partial plan P^0 relative to its external input. It is also possible that the plan size will stabilize as all inactive tokens are resolved and no new rules are fired. At each step the partial plan is evaluated to determine if it is complete. This evaluation is referred to as *Flaw Identification*. If no flaws are found, the plan is complete. If flaws are found, a single flaw is selected and resolved, resulting in a new partial plan. There are 3 types of flaws:

1. **Unbound Variable**: a variable in the partial plan whose domain is not a singleton. Unbound Variables are resolved by specification of a value from the domain of the variable.

2. **Open Condition**: an open condition is an inactive token. The operations to resolve an open condition are derived directly from the state model presented in Figure 4 on the Planning Representation page (i.e. merge, activate, and reject).
3. **Threat**: once a token has been placed in the partial plan it may impact other tokens indirectly through possible overlapping requirements on objects. Recall for example that a token may belong to objects (e.g. *Timelines*) which require a total order over their tokens. If any 2 tokens could possibly overlap (though not necessarily), then they pose a threat to each other in terms of achieving an extension of the current partial plan which is complete and consistent. Similarly, threats may arise where tokens share a common resource and their current state might yield extensions of the current partial plan which are inconsistent. Threats are resolved by imposing ordering constraints among tokens.

Open Conditions and Threats allow flaw detection and resolution at a higher-level of abstraction (i.e. in terms of objects and tokens) than that of simply binding variables as is common in Constraint Satisfaction Problems. This is advantageous when one applies heuristics for ordering choices since it provides a richer context in which to make decisions. Furthermore, it aids in reducing the amount of work done by a solver so that only the necessary refinements are made, otherwise leaving the partial plan with maximum flexibility. For example, one can omit unbound variables which are time-points of tokens since threats will force a solver to impose restrictions on these variables based on the semantics of the objects to which their tokens apply. Thus the planning process may yield partially-ordered plans for which all possible extensions are provably valid.

Refinement Search

The planning process depicted in Figure 1 is idealized in its assumption that all resolutions work first time. In practice, operations of refinement are interleaved with constraint propagation to test each partial plan for consistency. Inconsistent partial-plans, or partial-plans which include flaws that have no method of resolution, are dead-ends. This results in *search*. There is a wide-array of search algorithms available in industry and academia. The principal refinement search algorithm employed in EUROPA is a very common and very simple algorithm known as *chronological backtracking*. It is outlined in Figure 2. While other algorithms have and will be developed for EUROPA, this algorithm is the cornerstone of the search control framework provided in the Solver module of EUROPA.

```
// Function to solve a partial plan by successive refinement
// until all flaws are resolved.
bool solve(PartialPlan p){
  // Propagate the constraints to test for inconsistency. If found to be
  // inconsistent, then we can return false since this is a dead-end i.e. no refinements
  // to p can yield a consistent plan.
  if(isInconsistent(p))
    return false;

  // Non-deterministically choose a flaw from the set of available flaws.
  Flaw f = chooseFlaw(p);

  // If there are no flaws, then p is complete and we can terminate with success.
  if(f == NULL)
    return true;

  // Otherwise we formulate a decision point which is a branch in the search space. Each
  // choice is a particular refinement operation and the DecisionPoint collects all possible
  // refinement operations for the given flaw.
  DecisionPoint d = makeDecisionPoint(f, p);

  // Continue as long as we have something to try
  while(d.hasNext()){
```

```
    // A new partial plan is obtained by application of a refinement operator. Note that the ordering
    // over refinement operators to select is a non-deterministic step.
    PartialPlan pp = d.executeNext();

    // Recursive call to solve the new planning problem. If successful, then we are done.
    if(solve(pp))
      return true;
    else // Otherwise, retract the last refinement operation
      d.undo();
  }

  // If we arrive here, then we have exhausted all options to resolve the flaw, including the case wher
  // no options were available initially. Thus the problem cannot be solved.
  return false;
}
```

Figure 2: Chronological Backtracking Algorithm for Refinement Search

The algorithm takes as input a partial plan p and returns true if a complete and consistent refinement of p could be found (or if p is initially complete and consistent), and false otherwise. This algorithm provides for a sound and complete search, assuming that no flaws or available refinement operators are pruned unnecessarily. Dead-ends in the search are discovered through constraint propagation. Constraint propagation is a vehicle for evaluating the consistency of a partial plan and also for filtering infeasible values from consideration prior to commitment, allowing in some cases a strong look-ahead capability which is essential for tractable search. Consistency testing is initiated by the *isConsistent* procedure. The algorithm permits a heuristically controlled search by applying orderings for *chooseFlaw* and *makeDecisionPoint*. It results in a chronologically-backtracking, search. It should be emphasized that while this algorithm is commonly employed, it is only one of many that could be implemented with EUROPA.

---

1. Timeline instances are still objects, though they are objects of classes derived from a Timeline class.
2. A formal presntation of completeness is provided in (??).